Asynchronous Programming

# Exercise 01: setTimeOut

The setTimeout() function is commonly used if you wish to run your function a specified number of milliseconds from when the setTimeout() method was called. The general syntax of the method is:

|  |
| --- |
| setTimeout(expression, timeout); |

where expression is the JavaScript code to run after timeout milliseconds have elapsed.

Use **setTimeOut** to log **Hello Fresher Academy** after 1000ms

# Exercise 02: setInterval

The setInterval() function, as the name suggests is commonly used to set a delay for functions that are executed again and again like animations. The setInterval() function is very closely related to setTimeout() - they even have same syntax:

|  |
| --- |
| setInterval(expression, interval); |

The only difference is:

setTimeout()*triggers the*expression*only once while* setInterval()*keeps triggering*expression*regularly after the given interval of time. (unless you tell it to stop).*

Use **setInterval** to log **Hello Fresher Academy repeatedly after every 3 seconds**

# Exercise 03: Event Loop

Given the following code, what will be the output ?

|  |
| --- |
| (function() {  console.log(1);  setTimeout(function(){console.log(2)}, 1000);  setTimeout(function(){console.log(3)}, 0);  console.log(4);  })(); |

# Exercise 04: Event Loop cont

What order log messages take in the following cases?

## Case 1

|  |
| --- |
| console.log("A");  setImmediate(function() { console.log("B"); }, 0);  setImmediate(function() { console.log("C"); }, 0)  console.log("D"); |

## Case 2

|  |
| --- |
| setTimeout(function() {  setTimeout(function() {  console.log('A');  }, 0);  }, 0);  setTimeout(function() {  console.log('B');  }, 0);  setTimeout(function() {  setTimeout(function() {  console.log('C');  }, 0);  }, 10);  setTimeout(function() {  console.log('D');  }, 0); |

## Case 3

|  |
| --- |
| var x = 'A';  setTimeout(function() {  console.log(x);  x = 'B';  }, 3);  setTimeout(function() {  console.log(x);  x = 'C';  }, 2);  setTimeout(function() {  console.log(x);  x = 'D';  }, 1);  setTimeout(function() {  console.log(x);  }, 4); |

## Case 4

|  |
| --- |
| var t1 = setTimeout(function() {  console.log('A');  setTimeout(function() {  console.log('B');  }, 0);  }, 100);  var t2 = setTimeout(function() {  console.log('C');  setTimeout(function() {  console.log('D');  }, 0);  }, 200);  clearTimeout(t1);  setTimeout(function() {  clearTimeout(t2);  }, 250); |

# Exercise 05: Sync vs Async

Async calls, just as sync, can be structured and composed differently. You should be able to predict the sequence precisely without even running the code (running it in your mind). Always think about frames (event loop iterations) in which a particular code has to be executed.

What order log messages take in the following cases?

## Case 1

|  |
| --- |
| function logA() {  console.log('A');  }  function logB() {  console.log('B');  }  function logC() {  console.log('C');  }  function logD() {  console.log('D');  }  logD(logA(logB(logC()))); |

## Case 2

|  |
| --- |
| function alogA() {  setTimeout(function() {  console.log('A');  }, 0);  }  function alogB() {  setTimeout(function() {  console.log('B');  }, 0);  }  function alogC() {  setTimeout(function() {  console.log('C');  }, 0);  }  function alogD() {  setTimeout(function() {  console.log('D');  }, 0);  }  alogD(alogA(alogB(alogC()))); |

## Case 3

|  |
| --- |
| setTimeout(function() {  console.log('A');  setTimeout(function() {  console.log('B');  setTimeout(function() {  console.log('C');  setTimeout(function() {  console.log('D');  }, 0);  }, 100);  }, 200);  }, 300); |

## Case 4

|  |
| --- |
| setTimeout(function() {  console.log('A');  setTimeout(function() {  console.log('B');  }, 100);  }, 200);  setTimeout(function() {  console.log('C');  setTimeout(function() {  console.log('D');  }, 200);  }, 100); |

# Exercise 06: setTimeOut and Closure

Given the following code, what will be the output ?

|  |
| --- |
| for (var i = 0; i < 5; i++) {      setTimeout(function() { console.log(i); }, i \* 1000 );  } |

# Exercise 07: Tracking the scalpel

The village crows own an old scalpel that they occasionally use on special missions—say, to cut through screen doors or packaging. To be able to quickly track it down, every time the scalpel is moved to another nest, an entry is added to the storage of both the nest that had it and the nest that took it, under the name "scalpel", with its new location as the value.

This means that finding the scalpel is a matter of following the breadcrumb trail of storage entries, until you find a nest where that points at the nest itself.

Write an async function locateScalpel that does this, starting at the nest on which it runs. You can use the anyStorage function defined earlier to access storage in arbitrary nests. The scalpel has been going around long enough that you may assume that every nest has a "scalpel" entry in its data storage.

Next, write the same function again without using async and await.

Do request failures properly show up as rejections of the returned promise in both versions? How?

|  |
| --- |
| async function locateScalpel(nest) {  // Your code here.  }  function locateScalpel2(nest) {  // Your code here.  }  locateScalpel(bigOak).then(console.log);  // → Butcher Shop |

# Exercise 08: Building Promise.all

Given an array of promises, Promise.all returns a promise that waits for all of the promises in the array to finish. It then succeeds, yielding an array of result values. If a promise in the array fails, the promise returned by all fails too, with the failure reason from the failing promise.

Implement something like this yourself as a regular function called Promise\_all.

Remember that after a promise has succeeded or failed, it can’t succeed or fail again, and further calls to the functions that resolve it are ignored. This can simplify the way you handle failure of your promise.

|  |
| --- |
| function Promise\_all(promises) {  return new Promise((resolve, reject) => {  // Your code here.  });  }  // Test code.  Promise\_all([]).then(array => {  console.log("This should be []:", array);  });  function soon(val) {  return new Promise(resolve => {  setTimeout(() => resolve(val), Math.random() \* 500);  });  }  Promise\_all([soon(1), soon(2), soon(3)]).then(array => {  console.log("This should be [1, 2, 3]:", array);  });  Promise\_all([soon(1), Promise.reject("X"), soon(3)])  .then(array => {  console.log("We should not get here");  })  .catch(error => {  if (error != "X") {  console.log("Unexpected failure:", error);  }  }); |